Capítulo 4

Definição de Tipos

Além dos tipos pré-definidos no `Prelude` é possível definir novos tipos. O *Haskell* permite construir novos tipos através das construções do produto, co-produto e exponenciação. Vejamos como proceder.

4.1 Definição por Enumeração (co-produto)

Através da declaração `data` podemos definir novos tipos. Uma das formas de o fazer é através da enumeração dos vários elementos que vão constituir o novo tipo. Por exemplo um tipo que permite a classificação dos triângulos através da relação entre os comprimentos das suas três arestas.

```
data Triângulo = NãoÉTriângulo | Escaleno | Isóscele | Equilátero
```

Estamos a definir um novo tipo, `Triângulo`, através da construção de um co-produto ("`|`"), definido pelos construtores 0-ários `NãoÉTriângulo`, ..., `Equilátero`. O novo tipo só vai ter cinco valores, quatro deles definidos pelos construtores, e $\perp$, valor que está presente em todos os tipos *Haskell*. O novo tipo não possui nenhuma operação.

Podemos agora definir uma função que, dados três inteiros x, y e z, dados por ordem crescente, classifique o triângulo formado pelas arestas de comprimento x, y e z.

```
classificar :: (Int,Int,Int) -> Triângulo
classificar (x,y,z)
   | x+y<z = NãoÉTriângulo
   | x==z   = Equilátero
   | (x==y) || (y==z) = Isóscele
   | otherwise = Escaleno
```

De notar a utilização do valor lógico `otherwise` ($\doteq$ `True`) como forma de simplificar a escrita do último caso. No entanto se se tentar utilizar esta função vamos obter o seguinte resultado.

```
Main> classificar(1,2,3)
ERROR: Cannot find "show" function for:
*** expression : classificar (1,2,3)
*** of type    : Triângulo
```

[1] (Versão 1.3)

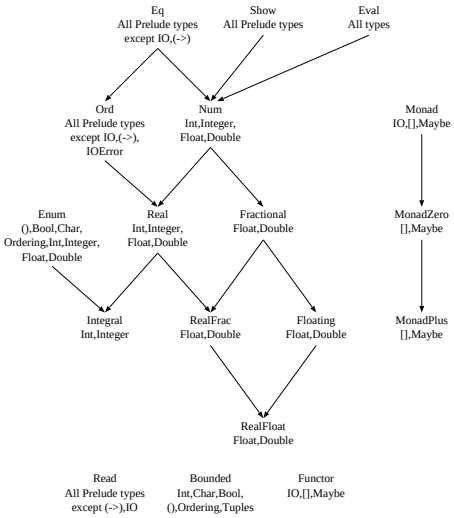
Traduzindo: "não consigo achar uma função para visualizar os valores do tipo Triângulo". Ou seja o `hugs` necessita de uma função de visualização para cada um dos seus tipos, ao definir-se um novo tipos essa função está em falha, veremos de seguida como proceder.

4.2 Classes

Em *Haskell* é possível definir novos tipos e inseri-los na estrutura de classe de tipos. Desta forma é possível obter funções para comparar valores dos novos tipos, funções para visualizar os novos valores, entre outras.

A estrutura de classe do *Haskell* dá conta das propriedades comuns aos diferentes tipos; por exemplo se os tipos admitem a comparação entre os seus membros, se admitem uma ordenação dos seus membros, entre outras.

A estrutura de classes do *Haskell* é a seguinte:



Tomando o exemplo do tipo `Triângulo`, vejamos como incorporar este novo tipo na estrutura já existente.

Pretende-se ter acesso às funções de visualização e de comparação.

A classe dos tipos para os quais é possível testar a igualdade entre os seus elementos é a classe `Eq`. Para incorporar o novo tipo nesta classe é necessário criar uma nova instância da classe e definir explicitamente o operador de igualdade para os elementos do novo tipo.

```
instance Eq Triângulo where
     NãoÉTriângulo == NãoÉTriângulo = True
     NãoÉTriângulo == Escaleno = False
     NãoÉTriângulo == Isóscele = False
     NãoÉTriângulo == Equilátero = False
     Escaleno == NãoÉTriângulo = False
     Escaleno == Escaleno = True
     Escaleno == Isóscele = False
     Escaleno == Equilátero = False
     Isóscele == NãoÉTriângulo = False
```

```
Isóscele == Escaleno = False
Isóscele == Isóscele = True
Isóscele == Equilátero = False
Equilátero == NãoÉTriângulo = False
Equilátero == Escaleno = False
Equilátero == Isóscele = False
Equilátero == Equilátero = True
```

A definição da relação `=/=`, não é necessário dado que esta é definida na classe `Eq` à custa da relação `==`. A definição da relação de igualdade desta forma é fastidiosa e, para tipos com mais elementos, rapidamente se torna impraticável.

Podemos usar a estrutura de classes e os operadores definidos nas diferentes classes como forma de obter uma definição mais compacta. Para tal vamos começar por estabelecer o tipo `Triângulo` como uma instância da classe `Enum`, a classe dos tipos enumeráveis.

```
instance Enum Triângulo where
  fromEnum NãoÉTriângulo = 0
  fromEnum Escaleno = 1
  fromEnum Isóscele = 2
  fromEnum Equilátero = 3
  toEnum 0 = NãoÉTriângulo
  toEnum 1 = Escaleno
  toEnum 2 = Isóscele
  toEnum 3 = Equilátero
```

Podemos agora aproveitar o facto de o tipo `Int` ser uma instância das classes `Eq` e `Ord` (entre outras), e as definições de `fromEnum` e de `toEnum`, para definir o tipo `Triângulo` como uma instância das classe `Eq` e `Ord` de uma forma simplificada.

```
instance Eq Triângulo where
  (x == y) = (fromEnum x == fromEnum y)


instance Ord Triângulo where
  (x < y) = (fromEnum x < fromEnum y)
```

No entanto, e dado que todas estas definições são óbvias, seria de esperar que o *Haskell* providenciasse uma forma automática de estabelecer estas instância de um tipo definido por enumeração. Felizmente tal é possível através da seguinte definição definição do tipo `Triângulo`.

```
data Triângulo = NãoÉTriângulo | Escaleno | Isóscele | Equilátero
  deriving (Eq,Enum,Ord,Show)
```

Com esta definição são criadas, automaticamente, instâncias do novo tipo nas classes `Eq`, `Enum`, `Ord` e `Show`.

Podemos então avaliar as seguintes expressões:

```
Main> classificar(4,4,4)
Equilátero
Main> classificar(4,4,4)==classificar(2,2,5)
False
```

4.3 Definição Paramétrica

A definição de novos tipos passa não só pela construção de co-produtos com construtores 0-ários, mas também pela construção de co-produtos, produtos e exponenciações com construtores n-ários. Por exemplo:

```
data Complexos = Par (Float,Float)
  deriving (Eq,Show)
```

define o tipo `Complexo` com um construtor binário que constrói um complexo como um par de reais.

De uma forma mais geral podemos definir tipos paramétricos, isto é tipos em cuja definição entram variáveis de tipo, e que desta forma não definem um só tipo mas um "classe" de tipos. Por exemplo, o tipo `Pares` pode ser definido do seguinte modo:

```
data Pares a b = ConstPares (a,b)
 deriving (Eq,Show)
```

com a definição das funções de projecção a serem definidas através da associação de padrões dados pelo construtor de pares `ConstPares`.

```
projX :: Pares a b -> a
projX (ConstPares (x,y)) = x

projY :: Pares a b -> b
projY (ConstPares (x,y)) = y
```

4.4 Definição Recursiva

É possível extender as definições de tipos ao caso das definições recursivas. Por exemplo o tipo `Lista` pode ser definido do seguinte modo:

```
data Lista a = Vazia | Lst (a,Lista a)
    deriving (Eq,Show)
```

Nesta caso temos dois construtores: `Vazia`, um construtor 0-ário, e `Lst` um construtor binário.

As definições de funções sobre elementos deste novo tipo podem ser definidas através de associação de padrões, tendo em conta os dois construtores disponibilizados.

Por exemplo a função que calcula o comprimento de uma lista.

```
comprimento :: (Lista a) -> Int
comprimento Vazia = 0
comprimento (Lst(x,l)) = 1+comprimento(l)
```

como exemplo de aplicação desta função temos:

```
Main> comprimento(Lst(1,Lst(2,Lst(3,Vazia))))
3
```

Outro exemplo de uma definição paramétrica e recursiva é nos dada pela definição de árvores binárias.

```
data ArvBin a = Folha a | Ramo (ArvBin a,ArvBin a)
 deriving (Show)
```

A definição da função de travessia `inordem` é nos dada por:

```
inordem :: (ArvBin a) -> [a]
inordem (Folha x) = [x]
inordem (Ramo(ab1,ab2)) = (inordem ab1 ) ++ (inordem ab2)
```

um exemplo de utilização é o seguinte:

```
Main> inordem(Ramo(Ramo(Folha 1,Folha 4),Folha 3))
[1, 4, 3]
```